LINKgroup

Distance Learning System

MySql

Indeksi, pogledi, uskladištene rutine i transakcije

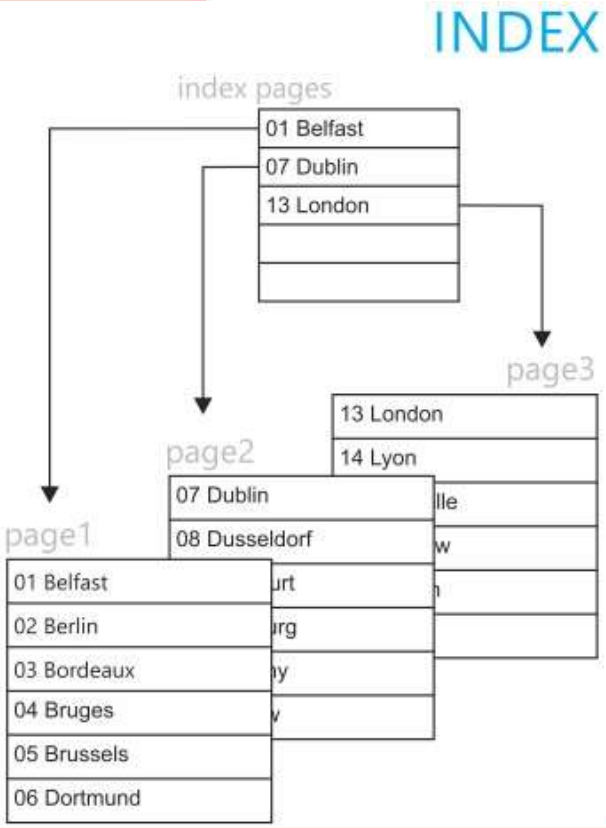
Indeksi

- U rukovanju bazama podataka, ključna stvar je brzo i precizno dobavljanje podataka. Kada su količine podataka u tabelama male, sistem će, u svakom slučaju, funkcionisati brzo. Ali, kada baza počne da sadrži nešto veće količine podataka, brzina postaje jedan od osnovnih problema. Međutim, nije brzina dostavljanja podataka ta koja proces čini sporim. Reč je o samom pronalaženju podataka. Kada se podatak jednom locira, brzina njegovog dostavljanja je irelevantna

1	Paris
2	Bruges
3	London
4	Moscow
5	Krakow

Mehanizam indeksa

- Većina podataka u MySQL bazi pamti na stranama. Veličina jedne strane je ograničena memorijski (obično su 8k ili 16k), ali ne može se tačno znati koliko u tu stranu može stati redova jer to zavisi od raznih faktora. Pre svega, zavisi od količine podataka u redovima.
- Ovako izgledaju strane za tabelu sa gradovima



Klasterovani i neklasterovani indeks

- Pretraga indeksa uvek funkcioniše na isti način, ali se njena finalizacija može izvršiti na dva načina. Ova dva načina čine da razlikujemo i dve vrste indeksa:
  - **klasterovani**
  - **neklasterovani (sekundarni)**
- Često se, sa punim pravom, ova dva indeksa porede sa knjigom i telefonskim imenikom. Ukoliko tražimo neku informaciju iz knjige, prvo ćemo pretražiti sadržaj i u njemu naći stranu za ono što nas zanima. Zatim ćemo prelistati knjigu i brzo doći do podatka, jer znamo stranu. Ako tražimo neki podatak u telefonskom imeniku, naći ćemo slovo koje nas zanima, a zatim direktno pristupiti strani na kojoj se nalaze svi podaci koji počinju tim slovom.
- Klasterovani indeks, na svom najnižem nivou (Leaf level) sadrži zapravo same podatke, dok neklasterovani indeks sadrži reference na prave podatke.
- Zato su neklasterovani indeksi nešto sporiji od klasterovanog

Formiranje Klasterovanog i neklasterovanog indeksa

- Svaki put kada kreiramo primarni ključ, automatski se kreira i klasterovani indeks. Ovaj indeks može biti samo jedan na nivou tabele
- Neklasterovane indekse kreiramo eksplicitno, i može ih biti više po jednoj tabeli

Strategija indeksiranja (klasterovani indeks)

- Kada napravimo jednu tabelu i dodelimo joj primarni ključ, MySQL server automatski, po toj koloni, kreira klasterovani indeks, pa je samo bitno da dobro isplaniramo šta će biti primarni ključ naše tabele, dok na sam klasterovani indeks i nećemo imati preveliki uticaj

```
create table mytable
(
id int primary key auto_increment,
name varchar(256)
);
```

```
show index from mytable
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type
mytable	0	PRIMARY	1	id	A	0	NULL	NULL		BTREE

Kreiranje neklasterovanog indeksa

- Neklasterovani indeks je malo pristupačniji i njega možemo postaviti gde god hoćemo.
- To ne znači da je pametno staviti ga na sve kolone tabele
- Indeks je, još jedna velika količina podataka koja će se naći na našem serveru. Takođe, to je i vlika količina podataka koje naš sistem treba da obrađuje.
- Indeksi su u stanju da znatno ubrzaju pretragu, ali, njima se takođe pristupa i prilikom drugih intervencija na tabelama.
- Što više indeksiranih kolona u tabeli, utoliko više usporenja uzrokovanih obradom indeksa prilikom rukovanja podacima te tabele.

```
ALTER TABLE mytable ADD INDEX custom_nonclustered_index (name ASC)
```

Full text index

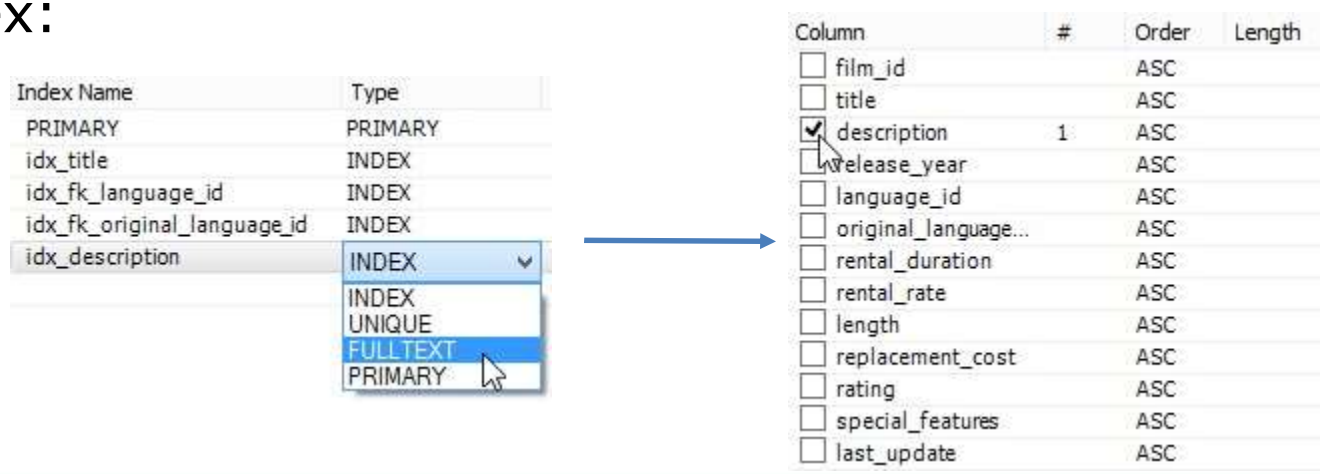
- Standardni indeks sa kojim smo se upoznali u dosadašnjem toku lekcije, kod tekstualnih kolona, sposoban je da prilikom pretrage uzme u razmatranje samo početne karaktere unosa. U situacijama kada u nekoj tekstualnoj koloni imamo zabeležen tekst koji se sastoji od više reči, običan indeks je praktično beskoristan.
- U takvim situacijama na scenu stupa Full-text Index.
- Full-text Index omogućava brzu pretragu kroz velike količine teksta. Ovo je dobar, funkcionalan, jednostavan i brz sistem i, što je najvažnije, nezamenljiv u nekim situacijama.
- Korišćenjem ovog indeksa, prilikom indeksiranja, indekser parsira tekst i preuzima iz njega reči. Svaka reč smešta se u indeks pod određenim brojem. Takođe, polje sa koga je tekst preuzet pamti se pod određenim brojem. Na kraju, indekser pravi relaciju između polja sa tekstom i rečima koje tom polju pripadaju.
- Kada dođe do pretrage, Engine prvo pronađe tražene reči, zatim izlista njihove relacije (u relacijama se pamti i broj pojavljivanja reči u polju), a zatim na osnovu tih relacija sortiranih po broju pojavljivanja reči, dobijamo i sama polja (odnosno, brojeve tih polja)

Kreiranje full text indeksa

- Ponekad ćemo pokušati da izvršimo pretragu na sledeći način:

```
select * from film where description like '%drama%'
```

- Kada kolone imaju mnogo teksta i ima mnogo redova u tabeli, ovo nije dobar koncept. Umesto njega, treba kreirati full text index:



Korišćenje full text indeksa

- Kada je full text indeks postavljen, postaju raspoložive i dodatne komande za pretragu:

```
SELECT * FROM film
WHERE MATCH(description) AGAINST('epic drama')
```

Filtracija rezultata (Search Expressions)

- Pored ovakvog standardnog načina definisanja upita za pretragu kolona korišćenjem Full Text Indexa, postoje i načini na koje možemo da utičemo na samu pretragu

```
SELECT description FROM film
WHERE MATCH(description) AGAINST('+epic +drama' IN BOOLEAN MODE);
```

```
SELECT description FROM film
WHERE MATCH(description) AGAINST('+epic -drama' IN BOOLEAN MODE);
```

Simbol	Značenje
+word	reč se mora pojaviti u rezultatu
-word	reč se ne sme pojaviti u rezultatu
~word	reč se može prijaviti u rezultatu, ali joj je data manja važnost
<word	reči se daje manje značenje prilikom pretrage
>word	reči se daje veće značenje prilikom pretrage
word*	sve reči koje počinju sa word (npr. word, words, wordless)
„word1 word2"	traži se fraza identična onoj navedenoj među navodnicima
()	zagrade se koriste za grupisanje izraza, npr. '+drama +(epic saga) ', tj. traže se svi izrazi koji sadrže fraze drama ili epic saga, ili i jedno i drugo

Pogledi

- Pogledi predstavljaju funkcionalnost SQL-a koja omogućava definisanje specijalne reprezentacije jedne ili više tabela.
- Ovo znači da je korišćenjem pogleda moguće kreirati takozvane virtualne tabele, koje mogu biti kombinacija kolona koje pripadaju različitim tabelama. Nad pogledima je zatim moguće vršiti SELECT upite i, u zavisnosti od definicije pogleda i prava korisnika, izmene podataka INSERT, UPDATE i DELETE naredbama.

Zašto koristimo poglede?

- **Dva najbitnija razloga korišćenja pogleda su sledeći:**
- **Sigurnost**
  - Veoma često može postojati situacija u kojoj želimo da određenim korisnicima baze podataka onemogućimo pun pristup određenoj tabeli ili tabelama.
  - Tipičan primer ove situacije su tabele za smeštanje podataka o zaposlenima. Mi ćemo svakako želeti da omogućimo svim zaposlenima pristup njihovim ličnim podacima, kao što su ime, prezime, broj telefona ili adresa. Ipak, zaposlenima nije dobro omogućiti pristup podacima o visini zarada i sličnim osetljivim podacima.
- **Komfor**
  - Već smo videli da je neophodno često vršiti spajanje podataka više tabela. Na primer, u bazi podataka Sakila, ukoliko bismo želeli da dobijemo kompletan set podataka o mušterijama, morali bismo da izvršimo upit kojim bismo spojili podatke nekoliko tabela. Kako bi se olakšao rad korisnicima baze podataka ili programerima koji će pisati aplikacije koje će komunicirati sa bazom, mi možemo olakšati stvari i kreirati pogled.

Kreiranje pogleda

- Pogledi (View) su objekti u bazi podataka. I zbog toga se, kao i ostali objekti, mogu kreirati DDL naredbom

```
CREATE VIEW myView AS query
```

```
create view uzmifilmove
as
select film.*,film_actor.actor_id from film_actor
join film on film_actor.film_id = film.film_id
```

Ažuriranje izvornih podataka

- Pogledi, osim mogućnosti prikazivanja aktuelnog sadržaja po zadatoj formi, imaju i mogućnost ažuriranja izvora na kojima su formirani, ali pri tom moraju biti ispoštovana neka pravila. Ova pravila se odnose na kreiranje SELECT upita prilikom definisanja pogleda. Ovo znači da se prilikom kreiranja pogleda pri pisanju SELECT upita moraju poštovati sledeća pravila:
- SELECT upit ne sme sadržati ključne reči GROUP BY, DISTINCT, LIMIT, UNION ili HAVING
- Pogledi koji grupišu podatke iz više tabela gotovo nikada se ne mogu koristiti za izmenu konkretnih podataka, tj. SELECT upit mora baratati samo jednom tabelom
- pogled mora sadržati sve kolone određene tabele koje su definisane kao primarni ili strani ključevi

Prava

- Pogledi su usko vezani sa bezbednošću i pravima korisnika. Ovi pojmovi će biti detaljno objašnjeni u sledećem modulu, ali ćemo i sada, ovde, napraviti kratku demonstraciju.
- Prilikom kreiranja pogleda, kreator pogleda mora imati prava nad tabelama nad kojima se pogled kreira. Sa druge strane, korisnik ne mora imati SELECT prava nad tabelama, sve dok ima prava nad pogledom.

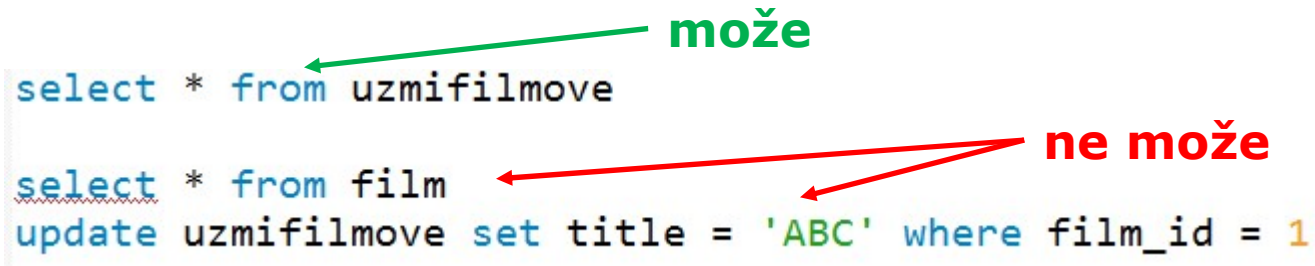
Vežba, kreiranje prava nad pogledom

- Kreirati korisnika John i dodeliti mu prava na pogled uzmifilmove

```
CREATE USER 'John'@'localhost' identified by '123'

grant select on uzmifilmove to John
```

- Ulogovati se kao John, a zatim isprobati sledeće upite:



Uskladištene rutine

- Uskladištene rutine su procedure i funkcije koje predstavljaju setove SQL naredbi koje su smeštene na serveru. Na ovaj način nije potrebno da klijenti ponavljaju određene naredbe više puta, već jednostavno mogu pozivati rutine
- Uskladištene rutine mogu biti posebno korisne u sledećim situacijama:
  - kada postoji više klijentskih aplikacija napisanih u različitim programskim jezicima, koje komuniciraju sa istom bazom podataka
  - kada je sigurnost imperativ; u bankarskim sistemima se na primer koriste uskladištene procedure i funkcije za sve operacije; ovo omogućava konzistentno i sigurno okruženje, a korišćenje rutina omogućava da je svaka operacija adekvatno logovana; u takvim scenarijima aplikacije i korisnici, nemaju direktan pristup tabelama baze podataka, već samo mogu da izvršavaju specifične uskladištene rutine

Stored procedures VS User-defined function

U sledećoj tabeli prikazane su osnovne razlike između procedura i funkcija

Stored procedures	**User-definied functions**
mogu vratiti null ili proizvoljan broj vrednosti	mogu vratiti samo jednu vrednost
mogu imati ulazne/uzlazne parametre	mogu imati samo ulazne parametre
moguće korišćenje SELECT i DML upita	moguće korišćenje samo SELECT upita
iz procedure mogu biti pozvane funkcije	iz funkcije ne može biti pozvana procedura
moguća obrada izuzetaka korišćenjem try-catch bloka	try-catch blokovi ne mogu biti korišćeni
ne mogu biti korišćene unutar SQL naredbi	mogu se koristiti unutar SQL naredbi
podržavaju transakcije	ne podržavaju transakcije

Uskladištene procedure

- Uskladištena (Stored) procedura je objekat na serveru, kao i svaki drugi. Stoga, da bismo je kreirali, koristimo DDL naredbu CREATE koja ima sledeću sintaksu:

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body
```

```
DELIMITER //
    CREATE PROCEDURE my_procedure()
        BEGIN
            SELECT 'hello from stored procedure';
        END//
DELIMITER ;
```

Strukturni parametri procedure

- Procedura može biti izgrađena sa nekolicinom strukturnih parametara (**CONTAINS SQL, NO SQL, READS SQL DATA, MODIFIES SQL DATA**) koje opisuju prirodu procedure, ali nemaju sintaksnog uticaja na naše rukovanje njome, već pomažu samom MySQL-u prilikom kreiranja statistika i optimizacije:

```
CREATE PROCEDURE my_procedure ()
MODIFIES SQL DATA
BEGIN
//procedure body
END;
```

Determinističke i nedeterminističke procedure

Sličnu svrhu ima i parametar DETERMINISTIC (i NON DETERMINISTIC). Ovaj parametar, takođe, veoma utiče na rad optimizacionog Enginea, ali nema preteran uticaj na naše sintaksne obaveze prilikom pisanja procedure. Ako je procedura označena kao deterministic, znači da se njen izlaz nikada ne menja, kao u prvoj proceduri koju smo napisali, a koja emituje poruku *hello from stored procedure*. Ako procedura nije deterministik (što je podrazumevani parametar), MySQL ne očekuje da rezultat procedure bude uvek isti (na primer, ako procedura sadrži funkciju now(), koja prikazuje tačno vreme i naravno, nikada ne daje identičan rezultat).

Parametrizacija procedure

- Procedura može prihvatiti parametre, i to:
  - Ulazne parametre **(in)**
  - Ulazno izlazne parametre **(inout)**
  - Izlazne parametre **(out)**
- Podrazumevana vrsta (kretanje) parametara procedure su ulazni parametri **(in)**

```
create procedure myproc(p1 int, p2 int)
select p1+p2
```

Parametrizacija procedure

- Osim in parametara, procedura može imati i **out** i **inout** parametre
- Oba se ponašaju kao reference (na primer objektni parametri u Javi)

```
delimiter //
create procedure proba(out p1 int)
begin
set p1 = 25;
end//
delimiter ;

call proba(@a);
select @a;
```

Parametrizacija procedure

- Razlika između out i inout je u tome što **out** nije u stanju da prihvati ulaznu vrednost parametra, a **inout** jeste

Ne može, vraća null

```
delimiter //
create procedure proba(out p1 int)
begin
set p1 = p1 + 1;
end//
delimiter ;

set @a=1;
call proba(@a);
select @a;
```

Može, vraća 2

```
delimiter //
create procedure proba(inout p1 int)
begin
set p1 = p1 + 1;
end//
delimiter ;

set @a=1;
call proba(@a);
select @a;
```

Vežba 1

- Kreirati proceduru za unos korisnika, sa nazivom usp_insertuser koja kao parametar prihvata ime korisnika. Korisnik će biti unet sa statusom user (broj 1)

Vežba 1 - rešenje

```
delimiter //
create PROCEDURE insertuser(newname varchar(50))
BEGIN
    insert into users (name,status) values (newname,1);
END//
```

Vežba 2

- Potrebno je napraviti uskladištenu proceduru koja kreira korisnika prema parametrizovanom imenu. Ukoliko korisnik sa tim imenom već postoji u tabeli, neće biti kreiran. Korisnik se unosi sa statusom user.

Vežba 2 - rešenje

```
delimiter //
create procedure insertuser(newname varchar(50))
begin
declare usersCount int;
select count(*) from users where username = newname into usersCount;
select usersCount;
if usersCount<1 then
    insert into users values (null,newname);
    select '0';
else
    select '-1';
end if;
end //
delimiter ;
```

Vežba 3

- Potrebno je napraviti proceduru koja će unositi korisnika u bazu. Procedura prihvata kao parametre ime i šifru korisnika. Ukoliko korisnik sa tim imenom ne postoji, uneće novog. Ukoliko korisnik postoji, biće mu zamenjena šifra novom. Korisnik se unosi sa statusom user.

Vežba 3 - rešenje

```
delimiter //
create PROCEDURE insertuser(newname varchar(50),newpass varchar(50))
BEGIN
   declare usersCount int;
   select count(id) from users where name=newname into usersCount;
   if usersCount<1 then
    insert into users (name,password,status) values (newname,password,1);
   else
    update users set password=newpass where name = newname;
   end if;
END//
delimiter ;
```

Vežba 4

- Napraviti proceduru koja će vratiti ime, prezime i ukupnu količinu potrošenog novca na porudžbine, na osnovu id-a korisnika prosleđenog kao parametar procedure

```
mysql> call usp_get_user_payments(15);
+------------+-----------+--------+
| first_name | last_name | am     |
+------------+-----------+--------+
| HELEN      | HARRIS    | 134.68 |
+------------+-----------+--------+
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

Vežba 4 rešenje

- Napraviti proceduru koja će vratiti ime, prezime i ukupnu količinu potrošenog novca na porudžbine, na osnovu id-a korisnika prosleđenog kao parametar procedure

```
delimiter //
drop procedure if exists usp_get_user_payments;
create procedure usp_get_user_payments(customer_id int)
begin

select first_name,last_name,sum(amount) am from payment
join customer on
payment.customer_id = customer.customer_id
where payment.customer_id = customer_id
group by first_name,last_name
order by am desc;

end//
delimiter ;
```

Vežba 5 - Money transfer functionality

U bazi podataka postoji sledeća tabela:

id int
username varchar
balance numeric(9,2)

Kreirati uskladištenu procuduru koja će prebacivati novac sa jednog na drugog korisnika (kolona balance)
Procedura treba da prihvata tri parametra:
Prvi parametar ce biti id korisnika kome će biti oduzet novac
Drugi parametar id korisnika kome se dodeljuje novac
Treći parametar će biti suma koja ce biti oduzeta od jednog i dodeljena drugom korisniku

Procedura mora proveriti da li korisnik ima dovoljno novca na računu pre nego što transakcija bude izvršena (balance izvornog korisnika ne sme da ode u minus)

Funkcije

- U MySql-u postoje dve vrste funkcija:
  - **Ugrađene**
  - **Korisnički definisane**
- Ugrađene funkcije su sve one funkcije koje podrazumeva standardna forma MySQL servera, dok su korisnički definisane one koje su izgrađene naknadno, od strane korisnika
- Sve funkcije se generalno dele na **skalarne** i **agregatne**

Korisnički definisane funkcije

- Korisnički definisane funkcije mogu biti realizovane na dva načina: kroz SQL skriptu (DDL) ili kroz izvorni programski jezik MySQL servera (C).
- Kreiranje korisnički definisanih funkcija (UDF) kroz SQL je jednostavniji metod. Zapravo, sintaksa je veoma slična sintaksi za kreiranje uskladištenih procedura.

Kreiranje funkcije

- Funkcija se kreira na isti način kao i procedura, ali je neophodno navesti njen izlazni tip prilikom kreiranja
- Takođe se za funkcije duže od jednog reda koristi zamena delimitera

```
CREATE FUNCTION myFunction()
RETURNS varchar(20)
RETURN 'hello from UDF';
```

```
DELIMITER //
CREATE FUNCTION myFunction()
RETURNS varchar(20)
BEGIN
DECLARE x int;
set x = 10;
RETURN 'pozdrav';
END //
DELIMITER ;
```

Pozivanje funkcije

- Za razliku od procedure, funkciju je moguće umetnuti u sam upit, što nam daje na raspolaganje velike mogućnosti za intervenciju na podacima u trenutku kreiranja izlaza. Na primer, funkciju pozivamo njenim umetanjem u upit:

```
SELECT myFunction()
```

Parametrizacija funkcije

- Parametrizacija funkcije vrši se na isti način kao i parametrizacija uskladištene procedure

```
delimiter //
create function myFunction(p1 int, p2 int)
returns int
begin
declare p3 int;
set p3 = p1 + p2;
return p3;
end //
delimiter ;
```

```
SELECT myFunction(2,3)
```

Vežba

- Potrebno je napraviti upit koji će prikazati glumce, ali tako da, svaki put kada se pojavi ime glumca christian, bude napisano neko drugo ime

Rešenje

- Ovaj problem se može rešiti funkcijom

```
delimiter //
create function changeName(p1 varchar(50), p2 varchar(50),p3 varchar(50))
returns varchar(50)
begin
    if p1=p2 then return p3; end if;
    return p1;
end //
delimiter ;
```

Vežba

- Potrebno je napraviti funkciju koja, za uneti naziv države vraća ukupan ukupnu zaradu iz te države

```
mysql> select udf_get_sum_for_country('yugoslavia');
+---------------------------------------+
| udf_get_sum_for_country('yugoslavia') |
+---------------------------------------+
|                                259.43 |
+---------------------------------------+
1 row in set (0.00 sec)
```

Rešenje

```
delimiter //
create function udf_get_sum_for_country(countryname varchar(256))
returns decimal(9,2)
deterministic
begin
return (select sum(payment.amount) as am from city
join address on address.city_id = city.city_id
join country on city.country_id = country.country_id
join customer on customer.address_id = address.address_id
join payment on payment.customer_id = customer.customer_id
where country.country like concat('%',countryname,'%')
group by country.country
order by am desc
limit 1);
end //
```

Vežba

- U bazi podataka se nalazi tabela ages_sms, koja prikazuje broj poslatih sms poruka u odnosu na starost korisnika
- Potrebno je napraviti funkciju koja će, na osnovu starosti novog korisnika, dati preporuku za paket (broj) sms poruka

```
mysql> select udf_predict_sms(40);
+---------------------+
| udf_predict_sms(40) |
+---------------------+
|               30.24 |
+---------------------+
1 row in set, 1 warning (0.00 sec)
```

Pomoć:

$$a = \frac{(\Sigma y\ \Sigma x^2) - (\Sigma x\ \Sigma xy)}{n(\Sigma x^2) - (\Sigma x)^2}$$

$$b = \frac{n(\Sigma xy) - (\Sigma x\ \Sigma y)}{n(\Sigma x^2) - (\Sigma x)^2}$$

$$\mathbf{y = a + (b * x)}$$

Rešenje

```
create function udf_predict_sms(userage int)
returns decimal(9,2)
deterministic
begin

return (
select
    -- a --
    (((sum(messages) * sum(pow(user_age,2))) - (sum(user_age) * sum(user_age * messages)))
    /
    ((count(*)*sum(pow(user_age,2))) - pow(sum(user_age),2)))
    +
    -- b --
    ((((count(*) * sum(user_age * messages)) - (sum(user_age) * sum(messages)))
    /
    ((count(*) * sum(pow(user_age,2))) - (pow(sum(user_age),2)))) * userage)
    from ages_sms

);
end//
```

Trigeri (okidači)

- Okidači su korisnički definisani blokovi koda koji se izvršavaju u trenutku intervencije na tabelama (insert, update i delete) i omogućavaju manipulisanje podacima pre nego što uistinu budu uneti, izmenjeni ili obrisani.
- Dele se na tri kategorije (**INSERT, UPDATE i DELETE**) i dve pod kategorije (**BEFORE i AFTER**) i u zavisnosti od tipa, nude određene opcije.
- Jedan okidač je objekat u jednoj bazi podataka i važi samo za tu bazu. Ukoliko se ta baza obriše, biće obrisani i svi njeni okidači.
- Kada kreiramo okidač, povezujemo ga sa određenom tabelom i akcijom koju želimo da pratimo. Pri tom, ne možemo uneti više istoimenih okidača u jednoj tabeli, niti postaviti iste tipove okidača na isti događaj u jednoj tabeli (na primer dva BEFORE INSERT okidača na jednoj tabeli)

Before triggers

- Karakteristika ovih okidača je da se događaju pre nego što podatak dođe do tabele. To nam omogućava da podatak presretnemo i izvršimo eventualnu intervenciju na njemu. U tu svrhu, MySQL u trenutku unosa kreira tabelu u memoriji, čija je struktura ista kao i ciljna tabela, s tom razlikom, što su jedini podaci ove tabele, u stvari, podaci koje unosimo ili menjamo:

```
create trigger checkName before insert on mytable
for each row set new.name = concat(new.name, "_test");
```

```
DELIMITER //
CREATE TRIGGER checkName BEFORE INSERT ON mytable
  FOR EACH ROW
    BEGIN
      if length(new.name) < 5 then
            set new.name = concat(new.name,"_test");
      END IF;
    END//
DELIMITER ;
```

```
SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT  = 'Ne moze';
```

Before triggers (update i delete)

- Update i delete trigeri funkcionišu isto kao i insert triger, samo se drugačije deklarišu

```
DELIMITER //
CREATE TRIGGER checkName BEFORE UPDATE ON mytable
 FOR EACH ROW
    BEGIN
      if length(old.name) < 4 then
            set new.name = password(new.name);
      END IF;
    END//
DELIMITER ;
```

After trigeri

- Ovi okidači aktiviraju se nakon unosa podataka. Znači da kada se nađemo u kodu okidača, imaćemo pristup samo već unetim podacima.
- *After Triggers* su dobri ukoliko želimo da paralelno sa glavnom tabelom ažuriramo i neku arhivsku, log ili bekap tabelu:

```
DELIMITER //
CREATE TRIGGER backup AFTER INSERT ON mytable
    FOR EACH ROW
        BEGIN
            insert into backup (name) values (new.name);
    END//
DELIMITER ;
```

After trigeri

- Kada jednom postavimo okidač na tabelu, onda je on deo te tabele u kontekstu njenih transakcija. To znači da se, prilikom ažuriranja podataka, uzima u obzir i izvršavanje okidača i da će se akcija smatrati neuspešnom ukoliko dođe do greške prilikom njegove aktivacije.
- Ukoliko dođe do greške prilikom aktivacije BEFORE okidača, neće doći ni do aktivacije AFTER okidača (ukoliko se nalaze na istoj naredbi iste tabele)

Kurzori

- Kurzori su posebni tipovi podataka koji omogućavaju sekvencijalno rukovanje podacima dobijenim određenim upitom. Iako kažemo tipovi podataka, rukovanje kurzorima nije baš jednostavno kao sa prostim tipovima. Oni imaju tabelarnu strukturu i zahtevaju specifičan set naredbi prilikom rukovanja.
- Drugim rečima kurzori se koriste kako bi se prošlo (iteriralo) kroz veliku kolekciju podataka. Pogledajmo na početku neke od osobenosti MySQL kurzora:
  - Nije moguće vršiti ažuriranje podataka korišćenjem kurzora, već samo čitanje
  - Kroz redove se može prolaziti samo u jednom smeru, i to tako kako je definisano SELECT upitom; nije moguće vršiti prolazak unazad ili slično
  - Kurzore je moguće koristiti samo unutar uskladištenih rutina (procedura i korisnički definisanih funkcija) i trigera

```
DELIMITER $$
CREATE PROCEDURE my_cursor()
BEGIN
 declare id int;
 declare fname varchar(50);
 DECLARE c CURSOR FOR SELECT actor_id,first_name FROM actor;
  OPEN c;
  REPEAT
   FETCH c INTO id,fname;
      if fname = 'CHRISTIAN' then
        update actor set first_name = 'CHRIS' where actor_id = id;
        end if;
    select fname;
   UNTIL isnull(id) END repeat;
  CLOSE c;
END
```

```
DELIMITER $$
CREATE PROCEDURE `film_cursor`()
BEGIN
 DECLARE sum float DEFAULT 0;
 DECLARE counter int DEFAULT 0;
 DECLARE a float;
 DECLARE e int default 0;
 DECLARE c CURSOR FOR SELECT length FROM film;

 DECLARE CONTINUE HANDLER FOR NOT FOUND SET e = 1;
  OPEN c;
  repeat
   FETCH c INTO a;
    IF NOT ISNULL(a) THEN
     SET sum = sum + a;
    END IF;

    SET counter = counter + 1;
   UNTIL e END repeat;
  CLOSE c;
 SELECT sum/counter;
END
```

Vežba

- Potrebno je napraviti tabelu bekap korisnika u koju će biti smešteni svi korisnici iz tabele users. Potrebno je napraviti logiku koja će da na svakog obrisanog ili unetog korisnika u tabeli users, reagovati tako što će istog korisnika obrisati ili uneti, u tabelu usersbackup.
- Pomoć (sintaksa za kreiranje insert i delete trigera):

```
delimiter //
create trigger setBackup after insert on users
  for each row
    begin
      ...
    end //
delimiter ;

delimiter //
create trigger deleteBackup after delete on users
  for each row
    begin
      ...
    end //
delimiter ;
```

Rešenje

```
create table usersbackup (id int, name varchar(50), password varchar(50),
status int);

delimiter //
create trigger setBackup after insert on users
  for each row
    begin
      insert into usersbackup (id,name,password,status)
      values (new.id,new.name,new.password,new.status);
end//
delimiter ;

      delimiter //
      create trigger deleteBackup after delete on users
        for each row
          begin
            delete from usersbackup where usersbackup.id = old.id;
          end//
      delimiter ;
```

Vežba

- Potrebno je napraviti before insert triger, koji će prilikom unosa korisnika istog imena u tabelu users svakom novom korisniku dodavati prefix existing.
- Pomoć:
- Kreiranje before insert trigera:

```
delimiter //
create trigger trg_checkUser before insert on users
  for each row
    begin
        ...
end//
delimiter ;
```

Rešenje

```
delimiter //
create trigger trg_checkUser before insert on users
  for each row
    begin
declare existing int;
      set existing = 0;
select count(id) from users where name = new.name into existing;
if existing > 0 then
set new.name = concat('existing_',new.name);
end if;
end//
delimiter ;
```